7

Delegates, Lambdas, and Events

WHAT'S IN THIS CHAPTER?

- Delegates
- Lambda expressions
- Closures
- Events

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/Delegates`.

The code for this chapter is divided into the following major examples:

- SimpleDelegates
- MulticastDelegates
- LambdaExpressions
- EventsSample

All the projects have nullable reference types enabled.

REFERENCING METHODS

Delegates are the .NET variant of addresses to methods. A delegate is an object-oriented type-safe pointer to one or multiple methods. Lambda expressions are directly related to delegates. When the parameter is a delegate type, you can use a lambda expression to implement a method that's referenced from the delegate.

This chapter explains the basics of delegates and lambda expressions, and it shows you how to implement methods called by delegates with lambda expressions. It also demonstrates how .NET uses delegates as the means of implementing events.

> **NOTE** *C# 9 also has the concept of a function pointer—a direct pointer to a managed or native method without the overhead of a delegate. Function pointers are explained in Chapter 13, "Managed and Unmanaged Memory."*

DELEGATES

In Chapter 4, "Object-Oriented Programming in C#," you read about using interfaces as contracts. If the parameter of a method has the type of an interface, with the implementation of the method any members of the interface can be used without being dependent on any interface implementation. Indeed, the implementation of the interface can be done independently of the method implementation. Similarly, a method can be declared to receive a parameter of a delegate type. The method receiving the delegate parameter can invoke the method that's referenced from the delegate. Similar to interfaces, the implementation of the method that's referenced by the delegate can be done independently of the method that's invoking the delegate.

The concept of passing delegates to methods can become clearer with some examples:

- **Tasks**—With tasks you can define a sequence of execution that should run in parallel with what currently is running in the main task. You can invoke the `Run` method of a `Task` and pass the address of a method via a delegate to invoke this method from the task. Tasks are explained in Chapter 11, "Tasks and Asynchronous Programming."
- **LINQ**—LINQ is implemented via extension methods that require a delegate as a parameter. Here you can pass functionality such as how to define the implementation to compare two values. LINQ is explained in detail in Chapter 9, "Language Integrated Query."
- **Events**—With events, you separate the producer that fires events and the subscribers that listen to events. The publisher and subscriber are decoupled. What's common between them is the contract of a delegate. Events are explained in detail later in this chapter.

Declaring Delegates

When you want to use a class in C#, you do so in two stages. First, you need to define the class—that is, you need to tell the compiler what fields and methods make up the class. Then (unless you are using only static methods), you instantiate an object of that class. With delegates, it is the same process. You start by declaring the delegates you want to use. Declaring delegates means telling the compiler what kind of method a delegate of that type will represent. Then, you have to create one or more instances of that delegate. Behind the scenes, a delegate type is a class, but there's specific syntax for delegates that hide details.

The syntax for declaring delegates looks like this:

```
delegate void IntMethodInvoker(int x);
```

This declares a delegate called `IntMethodInvoker` and indicates that each instance of this delegate can hold a reference to a method that takes one `int` parameter and returns `void`. The crucial point to understand about delegates is that they are type-safe. When you define the delegate, you have to provide full details about the signature and the return type of the method that it represents.

> **NOTE** *One good way to understand delegates is to think of a delegate as something that gives a name to a method signature and the return type.*

Suppose that you want to define a delegate called `TwoLongsOp` that represents a method that takes two `long`s as its parameters and returns a `double`. You could do so like this:

```
delegate double TwoLongsOp(long first, long second);
```

Or, to define a delegate that represents a method that takes no parameters and returns a `string`, you might write this (code file `GetAStringDemo/Program.cs`):

```
//...
delegate string GetAString();
```

The syntax is similar to that for a method definition, except there is no method body and the definition is prefixed with the keyword `delegate`. Because what you are doing here is basically defining a new class, you can define a delegate in any of the same places that you would define a class—that is to say, either inside another class, outside of any class, or in a namespace as a top-level object. Depending on how visible you want your definition to be and the scope of the delegate, you can apply any of the access modifiers that also apply to classes to define its visibility:

```
public delegate string GetAString();
```

> **NOTE** *Delegates are implemented as classes derived from the class* `System.MulticastDelegate`*, which is derived from the base class* `System.Delegate`*. The C# compiler is aware of this class and uses the delegate syntax to hide the details of the operations of this class.*

After you have defined a delegate, you can create an instance of it so that you can use it to store details about a particular method.

Using Delegates

The following code snippet demonstrates the use of a delegate. It is a rather long-winded way of calling the `ToString` method on an `int` (code file `GetAStringDemo/Program.cs`):

```
int x = 40;
GetAString firstStringMethod = new GetAString(x.ToString);
Console.WriteLine($"String is {firstStringMethod()}");
//...
```

This code instantiates a delegate of type `GetAString` and initializes it so it refers to the `ToString` method of the integer variable `x`. Delegates always take a one-parameter constructor, which is the address of a method. This method must match the signature and return type with which the delegate was defined. Because `ToString` is an instance method (as opposed to a static method), the instance needs to be supplied with the parameter.

The next line invokes the delegate to display the string. In any code, supplying the name of a delegate instance, followed by parentheses containing any parameters, has exactly the same effect as calling the method wrapped by the delegate.

In fact, supplying parentheses to the delegate instance is the same as calling the `Invoke` method of the delegate class. Because `firstStringMethod` is a variable of a delegate type, the C# compiler replaces `firstStringMethod` with `firstStringMethod.Invoke`:

```
firstStringMethod();
firstStringMethod.Invoke();
```

For less typing, at every place where a delegate instance is needed, you can just pass the name of the address. This is known by the term *delegate inference*. This C# feature works as long as the compiler can resolve the delegate instance to a specific type. The example initialized the variable `firstStringMethod` of type `GetAString` with a new instance of the delegate `GetAString`:

```
GetAString firstStringMethod = new GetAString(x.ToString);
```

You can write the same just by passing the method name with the variable `x` to the variable `firstStringMethod`:

```
GetAString firstStringMethod = x.ToString;
```

The code that is created by the C# compiler is the same. The compiler detects that a delegate type is required with `firstStringMethod`, so it creates an instance of the delegate type `GetAString` and passes the address of the method with the object `x` to the constructor.

> **NOTE** *Be aware that you can't add the parentheses to the method name as* `x.ToString()` *and pass it to the delegate variable. This would be an invocation of the method. The invocation of the* `ToString` *method returns a string object that can't be assigned to the delegate variable. You can only assign the address of a method to the delegate variable.*

Delegate inference can be used anywhere a delegate instance is required. Delegate inference can also be used with events because events are based on delegates (as you'll see later in this chapter).

One feature of delegates is that they are type-safe to the extent that they ensure that the signature of the method being called is correct. However, interestingly, they don't care what type of object the method is being called against or even whether the method is a static method or an instance method.

> **NOTE** *An instance of a given delegate can refer to any instance or static method on any object of any type provided that the signature of the method matches the signature of the delegate.*

To demonstrate this, the following example expands the previous code snippet so that it uses the `firstStringMethod` delegate to call a couple of other methods on another object—an instance method and a static method. For this, the `Currency` struct is defined. This type has its own overload of `ToString` and a static method with the same signature to `GetCurrencyUnit`. This way, the same delegate variable can be used to invoke these methods (code file `GetAStringDemo/Currency.cs`):

```
struct Currency
{
  public uint Dollars;
  public ushort Cents;
```

```
    public Currency(uint dollars, ushort cents)
    {
      Dollars = dollars;
      Cents = cents;
    }

    public override string ToString() => $"${Dollars}.{Cents,2:00}";

    public static string GetCurrencyUnit() => "Dollar";

    public static explicit operator Currency (float value)
    {
      checked
      {
        uint dollars = (uint)value;
        ushort cents = (ushort)((value — dollars) * 100);
        return new Currency(dollars, cents);
      }
    }

    public static implicit operator float (Currency value) =>
      value.Dollars + (value.Cents / 100.0f);

    public static implicit operator Currency (uint value) =>
      new Currency(value, 0);

    public static implicit operator uint (Currency value) =>
      value.Dollars;
  }
```

Now you can use the `GetAString` instance as follows (code file `GetAStringDemo/Program.cs`):

```
private delegate string GetAString();

//...
var balance = new Currency(34, 50);

// firstStringMethod references an instance method
firstStringMethod = balance.ToString;
Console.WriteLine($"String is {firstStringMethod()}");

// firstStringMethod references a static method
firstStringMethod = new GetAString(Currency.GetCurrencyUnit);
Console.WriteLine($"String is {firstStringMethod()}");
```

This code shows how you can call a method via a delegate and subsequently reassign the delegate to refer to different methods on different instances of classes, even static methods or methods against instances of different types of class, provided that the signature of each method matches the delegate definition.

When you run the application, you get the output from the different methods that are referenced by the delegate:

```
String is 40
String is $34.50
String is Dollar
```

Now that you've been introduced to the foundations of delegates, it's time to move onto something more useful and practical: passing delegates to methods.

Passing Delegates to Methods

This example defines a `MathOperations` class that uses a couple of static methods to perform two operations on doubles. Then you use delegates to invoke these methods. The `MathOperations` class looks like this (code file `SimpleDelegates/MathOperations`):

```
public static class MathOperations
{
  public static double MultiplyByTwo(double value) => value * 2;
  public static double Square(double value) => value * value;
}
```

You invoke these methods as follows (code file `SimpleDelegates/Program.cs`):

```
using System;

DoubleOp[] operations =
{
  MathOperations.MultiplyByTwo,
  MathOperations.Square
};

for (int i=0; i < operations.Length; i++)
{
  Console.WriteLine($"Using operations[{i}]");
  ProcessAndDisplayNumber(operations[i], 2.0);
  ProcessAndDisplayNumber(operations[i], 7.94);
  ProcessAndDisplayNumber(operations[i], 1.414);
  Console.WriteLine();
}

void ProcessAndDisplayNumber(DoubleOp action, double value)
{
  double result = action(value);
  Console.WriteLine($"Value is {value}, result of operation is {result}");
}

delegate double DoubleOp(double x);
```

In this code, you instantiate an array of `DoubleOp` delegates (remember that after you have defined a delegate class, you can basically instantiate instances just as you can with normal classes, so putting some into an array is no problem). Each element of the array is initialized to refer to a different operation implemented by the `MathOperations` class. Then, you loop through the array, applying each operation to three different values. This illustrates one way of using delegates—to group methods together into an array so that you can call several methods in a loop.

The key lines in this code are the ones in which you actually pass each delegate to the `ProcessAndDisplayNumber` method, such as this:

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

This passes in the name of a delegate but without any parameters. Given that `operations[i]` is a delegate, syntactically the following is true:

- `operations[i]` means the delegate (that is, the method represented by the delegate).
- `operations[i](2.0)` means actually calling this method, passing in the value in parentheses.

The ProcessAndDisplayNumber method is defined to take a delegate as its first parameter:

```
void ProcessAndDisplayNumber(DoubleOp action, double value)
```

Then, within the implementation of this method, you call this:

```
double result = action(value);
```

This actually causes the method that is wrapped up by the action delegate instance to be called, and its return result is stored in Result. Running this example gives you the following:

```
Using operations[0]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 15.88
Value is 1.414, result of operation is 2.828

Using operations[1]:
Value is 2, result of operation is 4
Value is 7.94, result of operation is 63.043600000000005
Value is 1.414, result of operation is 1.9993959999999997
```

> **NOTE** *With the outcome you're seeing, you might expect different results with some of the multiplications, but if you round the results, they match. This is because of how* double *values are stored. Depending on the data you're working with, this might not be good enough—for example with financial data. Here you should use the* decimal *type instead.*

Action<T> and Func<T> Delegates

Instead of defining a new delegate type with every parameter and return type, you can use the Action<T> and Func<T> delegates. The generic Action<T> delegate is meant to reference a method with void return. This delegate class exists in different variants so that you can pass up to 16 different parameter types. The Action class without the generic parameter is for calling methods without parameters. Action<in T> is for calling a method with one parameter; Action<in T1, in T2> is for a method with two parameters; and Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8> is for a method with eight parameters.

The Func<T> delegates can be used in a similar manner. Func<T> allows you to invoke methods with a return type. Similar to Action<T>, Func<T> is defined in different variants to pass up to 16 parameter types and a return type. Func<out TResult> is the delegate type to invoke a method with a return type and without parameters. Func<in T, out TResult> is for a method with one parameter, and Func<in T1, in T2, in T3, in T4, out TResult> is for a method with four parameters.

The example in the preceding section declared a delegate with a double parameter and a double return type:

```
delegate double DoubleOp(double x);
```

Instead of declaring the custom delegate DoubleOp, you can use the Func<in T, out TResult> delegate. You can declare a variable of the delegate type or, as shown here, an array of the delegate type:

```
Func<double, double>[] operations =
{
  MathOperations.MultiplyByTwo,
  MathOperations.Square
};
```

and use it with the ProcessAndDisplayNumber method as a parameter:

```
static void ProcessAndDisplayNumber(Func<double, double> action,
  double value)
```

```
{
  double result = action(value);
  Console.WriteLine($"Value is {value}, result of operation is {result}");
}
```

Multicast Delegates

So far, each of the delegates you have used wraps just one method call. Calling the delegate amounts to calling that method. If you want to call more than one method, you need to make an explicit call through a delegate more than once. However, it is possible for a delegate to wrap more than one method. Such a delegate is known as a *multicast delegate*. When a multicast delegate is called, it successively calls each method in order. For this to work, the delegate signature should return a `void`; otherwise, you would only get the result of the last method invoked by the delegate.

With a `void` return type, you can use the `Action<double>` delegate (code file `MulticastDelegates/Program.cs`):

```
Action<double> operations = MathOperations.MultiplyByTwo;
operations += MathOperations.Square;
```

In the earlier example, you wanted to store references to two methods, so you instantiated an array of delegates. Here, you simply add both operations into the same multicast delegate. Multicast delegates recognize the operators `+`, `+=`, and `-=`. Alternatively, you can expand the last two lines of the preceding code, as in this snippet:

```
Action<double> operation1 = MathOperations.MultiplyByTwo;
Action<double> operation2 = MathOperations.Square;
Action<double> operations = operation1 + operation2;
```

With the sample project `MulticastDelegates`, the `MathOperations` type from `SimpleDelegates` has been changed to return void and to display the results on the console (code file `MulticastDelegates/MathOperations.cs`):

```
public static class MathOperations
{
  public static void MultiplyByTwo(double value) =>
    Console.WriteLine($"Multiplying by 2: {value} gives {value * 2}");

  public static void Square(double value) =>
    Console.WriteLine($"Squaring: {value} gives {value * value}");
}
```

To accommodate this change, you also have to rewrite `ProcessAndDisplayNumber` (code file `MulticastDelegates/Program.cs`):

```
static void ProcessAndDisplayNumber(Action<double> action, double value)
{
  Console.WriteLine($"ProcessAndDisplayNumber called with value = {value}");
  action(value);
  Console.WriteLine();
}
```

Now you can try your multicast delegate:

```
Action<double> operations = MathOperations.MultiplyByTwo;
operations += MathOperations.Square;
ProcessAndDisplayNumber(operations, 2.0);
```

```
ProcessAndDisplayNumber(operations, 7.94);
ProcessAndDisplayNumber(operations, 1.414);
```

Each time `ProcessAndDisplayNumber` is called, it displays a message saying that it has been called. Then the following statement causes each of the method calls in the `action` delegate instance to be called in succession:

```
action(value);
```

Running the preceding code produces this result:

```
ProcessAndDisplayNumber called with value = 2
Multiplying by 2: 2 gives 4
Squaring: 2 gives 4

ProcessAndDisplayNumber called with value = 7.94
Multiplying by 2: 7.94 gives 15.88
Squaring: 7.94 gives 63.043600000000005

ProcessAndDisplayNumber called with value = 1.414
Multiplying by 2: 1.414 gives 2.828
Squaring: 1.414 gives 1.9993959999999997
```

If you are using multicast delegates, be aware that the order in which methods chained to the same delegate will be called is formally undefined. Therefore, avoid writing code that relies on such methods being called in any particular order.

Invoking multiple methods by one delegate might cause an even bigger problem. The multicast delegate contains a collection of delegates to invoke one after the other. If one of the methods invoked by a delegate throws an exception, the complete iteration stops. Consider the following `MulticastIteration` example. Here, the simple delegate `Action` is used. This delegate is meant to invoke the methods `One` and `Two`, which fulfill the parameter and return type requirements of the delegate. Be aware that method `One` throws an exception (code file `MulticastDelegatesUsingInvocationList/Program.cs`):

```
static void One()
{
  Console.WriteLine("One");
  throw new Exception("Error in One");
}

static void Two()
{
  Console.WriteLine("Two");
}
```

With the top-level statements, delegate `d1` is created to reference method `One`; next, the address of method `Two` is added to the same delegate. `d1` is invoked to call both methods. The exception is caught in a `try`/`catch` block:

```
Action d1 = One;
d1 += Two;
try
{
  d1();
}
catch (Exception)
{
  Console.WriteLine("Exception caught");
}
```

Only the first method is invoked by the delegate. Because the first method throws an exception, iterating the delegates stops here, and method `Two` is never invoked. The result might differ because the order of calling the methods is not defined:

```
One
Exception Caught
```

> **NOTE** *Errors and exceptions are explained in detail in Chapter 10, "Errors and Exceptions."*

In such a scenario, you can avoid the problem by iterating the list on your own. The `Delegate` class defines the method `GetInvocationList` that returns an array of `Delegate` objects. You can now use these delegates to invoke the methods associated with them directly, catch exceptions, and continue with the next iteration (code file `MulticastDelegatesUsingInvocationList/Program.cs`):

```
Action d1 = One;
d1 += Two;
Delegate[] delegates = d1.GetInvocationList();
foreach (Action d in delegates)
{
  try
  {
    d();
  }
  catch (Exception)
  {
    Console.WriteLine("Exception caught");
  }
}
```

When you run the application with the code changes, you can see that the iteration continues with the next method after the exception is caught:

```
One
Exception caught
Two
```

Anonymous Methods

Up to this point, a method must already exist for the delegate to work (that is, the delegate is defined with the same signature as the method(s) it will be used with). However, there is another way to use delegates—with *anonymous methods*. An anonymous method is a block of code that is used as the parameter for the delegate.

The syntax for defining a delegate with an anonymous method doesn't change. It's when the delegate is instantiated that things change. The following simple console application shows how using an anonymous method can work (code file `AnonymousMethods/Program.cs`):

```
string mid = ", middle part,";
Func<string, string> anonDel = delegate(string param)
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
Console.WriteLine(anonDel("Start of string"));
```

The delegate `Func<string, string>` takes a single string parameter and returns a string. `anonDel` is a variable of this delegate type. Instead of assigning the name of a method to this variable, a simple block of code is used, prefixed by the `delegate` keyword and followed by a string parameter.

As you can see, the block of code uses a method-level string variable, `mid`, which is defined outside of the anonymous method and adds it to the parameter that was passed in. The code then returns the string value. When the delegate is called, a string is passed in as the parameter, and the returned string is output to the console.

The benefit of using anonymous methods is that it reduces the amount of code you have to write. You don't need to define a method just to use it with a delegate. This becomes evident when you define the delegate for an event (events are discussed later in this chapter), and it helps reduce the complexity of the code, especially where several events are defined. With anonymous methods, the code does not perform faster. The compiler still defines a method; the method just has an automatically assigned name that you don't need to know.

You must follow a couple of rules when using anonymous methods. An anonymous method can't have a jump statement (`break`, `goto`, or `continue`) that has a target outside of the anonymous method. The reverse is also true: a jump statement outside the anonymous method cannot have a target inside the anonymous method.

If you have to write the same functionality more than once, don't use anonymous methods. In this case, instead of duplicating the code, write a named method. You have to write it only once and reference it by its name.

> **NOTE** *The syntax for anonymous methods was introduced with C# 2. With new programs, you really don't need this syntax anymore because lambda expressions (explained in the next section) offer the same—and more—functionality. However, you'll find the syntax for anonymous methods in many places in existing source code, which is why it's good to know it.*
>
> *Lambda expressions have been available since C# 3.*

LAMBDA EXPRESSIONS

One way lambda expressions are used is to assign code—using a lambda expression—to a parameter. You can use lambda expressions whenever you have a delegate parameter type. The previous example using anonymous methods is modified in the following snippet to use a lambda expression:

```
string mid = ", middle part,";
Func<string, string> lambda = param =>
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
Console.WriteLine(lambda("Start of string"));
```

The left side of the lambda operator, `=>`, lists the necessary parameters. The right side following the lambda operator defines the implementation of the method assigned to the variable `lambda`.

Parameters

With lambda expressions, there are several ways to define parameters. If there's only one parameter, just the name of the parameter is enough. The following lambda expression uses the parameter named `s`. Because the delegate type defines a `string` parameter, `s` is of type `string`. The implementation returns a formatted string that is finally written to the console when the delegate is invoked: `change uppercase TEST` (code file `LambdaExpressions/Program.cs`):

```
Func<string, string> oneParam = s => $"change uppercase {s.ToUpper()}";
Console.WriteLine(oneParam("test"));
```

If a delegate uses more than one parameter, you can combine the parameter names inside brackets. Here, the parameters `x` and `y` are of type `double` as defined by the `Func<double, double, double>` delegate:

```
Func<double, double, double> twoParams = (x, y) => x * y;
Console.WriteLine(twoParams(3, 2));
```

For convenience, you can add the parameter types to the variable names inside the brackets. If the compiler can't match an overloaded version, using parameter types can help resolve the matching delegate:

```
Func<double, double, double> twoParamsWithTypes =
  (double x, double y) => x * y;
Console.WriteLine(twoParamsWithTypes(4, 2));
```

Multiple Code Lines

If the lambda expression consists of a single statement, a method block with curly brackets and a return statement are not needed. There's an implicit `return` added by the compiler:

```
Func<double, double> square = x => x * x;
```

It's completely legal to add curly brackets, a `return` statement, and semicolons. Usually it's just easier to read without them:

```
Func<double, double> square = x =>
{
  return x * x;
};
```

However, if you need multiple statements in the implementation of the lambda expression, curly brackets and the `return` statement are required:

```
Func<string, string> lambda = param =>
{
  param += mid;
  param += " and this was added to the string.";
  return param;
};
```

Closures

With lambda expressions, you can access variables outside the block of the lambda expression. This is known as *closure*. Closures are a great feature, but they can also be dangerous if not used correctly.

In the following example, a lambda expression of type `Func<int, int>` requires one `int` parameter and returns an `int`. The parameter for the lambda expression is defined with the variable `x`. The implementation also accesses the variable `someVal`, which is outside the lambda expression. As long as you do not assume that the lambda expression creates a new method that is used later when `f` is invoked, this might not look confusing at all. Looking at this code block, the returned value calling `f` should be the value from `x` plus 5, but this might not be the case (code file `LambdaExpressions/Program.cs`):

```
int someVal = 5;
Func<int, int> f = x => x + someVal;
```

Assuming the variable `someVal` is later changed and then the lambda expression is invoked, the new value of `someVal` is used. The result of invoking `f(3)` is `10`:

```
someVal = 7;
Console.WriteLine(f(3));
```

Similarly, when you're changing the value of a closure variable within the lambda expression, you can access the changed value outside of the lambda expression.

Now, you might wonder how it is possible at all to access variables outside of the lambda expression from within the lambda expression. To understand this, consider what the compiler does when you define a lambda expression. With the lambda expression `x => x + someVal`, the compiler creates an anonymous class that has a constructor to pass the outer variable. The constructor depends on how many variables you access from the outside. With this simple example, the constructor accepts an `int`. The anonymous class contains an anonymous method that has the implementation as defined by the lambda expression, with the parameters and return type:

```
public class AnonymousClass
{
  private int _someVal;
  public AnonymousClass(int someVal) => _someVal = someVal;

  public int AnonymousMethod(int x) => x + _someVal;
}
```

In case a value outside of the scope of the lambda expression needs to be returned, a reference type is used.

Using the lambda expression and invoking the method creates an instance of the anonymous class and passes the value of the variable from the time when the call is made.

> **NOTE** *In case you are using closures with multiple threads, you can get into concurrency conflicts. It's best to use only immutable types for closures. This way it's guaranteed the value can't change, and synchronization is not needed.*

> **NOTE** *You can use lambda expressions anywhere the type is a delegate. Another use of lambda expressions is when the type is* `Expression` *or* `Expression<T>`*, in which case the compiler creates an expression tree. This feature is discussed in Chapter 9.*

EVENTS

Events are based on delegates and offer a publish/subscribe mechanism to delegates. You can find events everywhere across the framework. In Windows applications, the `Button` class offers the `Click` event. This type of event is a delegate. A handler method that is invoked when the `Click` event is fired needs to be defined and to include parameters as defined by the delegate type.

> **NOTE** *See design guidelines for events in the Microsoft documentation:* `https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/event`.

In the code example shown in this section, events are used to connect the `CarDealer` and `Consumer` classes. The `CarDealer` class offers an event when a new car arrives. The `Consumer` class subscribes to the event to be informed when a new car arrives.

Event Publisher

You start with a `CarDealer` class that offers a subscription based on events. `CarDealer` defines the event named `NewCarCreated` of type `EventHandler<CarInfoEventArgs>` with the `event` keyword. Inside the method `CreateANewCar`, the event `NewCarCreated` is fired by invoking the method `RaiseNewCarCreated`. The implementation of this method verifies whether the delegate is not null and raises the event (code file `EventsSample/CarDealer.cs`):

```
public class CarInfoEventArgs: EventArgs
{
  public CarInfoEventArgs(string car) => Car = car;
  public string Car { get; }
}

public class CarDealer
{
  public event EventHandler<CarInfoEventArgs>? NewCarInfo;
  public void CreateANewCar(string car)
  {
    Console.WriteLine($"CarDealer, new car {car}");
    RaiseNewCarCreated(car);
  }

  private void RaiseNewCarCreated(string car) =>
    NewCarCreated?.Invoke(this, new CarInfoEventArgs(car));
}
```

The class `CarDealer` offers the event `NewCarCreated` of type `EventHandler<CarInfoEventArgs>`. As a convention, events typically use methods with two parameters; the first parameter is an object and contains the sender of the event, and the second parameter provides information about the event. The second parameter is different for various event types. You could create a specific delegate type such as

```
public delegate void NewCarCreatedHandler(object sender, CarInfoEventArgs e);
```

or use the generic type `EventHandler` as shown in the sample code. With `EventHandler<TEventArgs>`, the first parameter needs to be of type `object`, and the second parameter is of type `T`. `EventHandler<TEventArgs>` also defines a constraint on `T`; it must derive from the base class `EventArgs`, which is the case with `CarInfoEventArgs`.

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

The delegate `EventHandler<TEventArgs>` is defined as follows:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
  where TEventArgs: EventArgs
```

Defining the event in one line is a C# shorthand notation. The compiler creates a variable of the delegate type `EventHandler<CarInfoEventArgs>` and adds methods to subscribe and unsubscribe from the delegate. The long form of the shorthand notation is shown next. This is similar to auto-properties and full properties. With events, the `add` and `remove` keywords are used to add and remove a handler to the delegate:

```
private EventHandler<CarInfoEventArgs>? _newCarCreated;
public event EventHandler<CarInfoEventArgs>? NewCarCreated
{
  add => _newCarCreated += value;
  remove => _newCarCreated -= value;
}
```

> **NOTE** *The long notation to define events is useful if more needs to be done than just adding and removing the event handler, such as adding synchronization for multiple thread access. The UWP, WPF, and WinUI controls make use of the long notation to add bubbling and tunneling functionality with the events.*

The class `CarDealer` fires the event by calling the `Invoke` method of the delegate. This invokes all the handlers that are subscribed to the event. Remember, as previously shown with multicast delegates, the order of the methods invoked is not guaranteed. To have more control over calling the handler methods, you can use the `Delegate` class method `GetInvocationList` to access every item in the delegate list and invoke each on its own, as shown earlier.

```
NewCarCreated?.Invoke(this, new CarInfoEventArgs(car));
```

Firing the event requires only a one-liner. Prior to C# 6, firing the event was more complex—checking the delegate for `null` (if no subscriber was registered) before invoking the method, which should have been done in a thread-safe manner. Now, checking for null is done using the `?.` operator.

Event Listener

The class `Consumer` is used as the event listener. This class subscribes to the event of the `CarDealer` and defines the method `NewCarIsHere` that in turn fulfills the requirements of the `EventHandler<CarInfoEventArgs>` delegate with parameters of type `object` and `CarInfoEventArgs` (code file `EventsSample/Consumer.cs`):

```
public record Consumer(string Name)
{
  public void NewCarIsHere(object? sender, CarInfoEventArgs e) =>
    Console.WriteLine($"{Name}: car {e.Car} is new");
}
```

Now the event publisher and subscriber need to connect. You do this by using the `NewCarInfo` event of the `CarDealer` to create a subscription with `+=`. The consumer `sebastian` subscribes to the event, and after the car Williams is created, the consumer `max` subscribes. After the car Aston Martin is created, `sebastian` unsubscribes with `-=` (code file `EventsSample/Program.cs`):

```
CarDealer dealer = new();
Consumer sebastian = new("Sebastian");
dealer.NewCarInfo += sebastian.NewCarIsHere;
dealer.NewCar("Williams");

Consumer max = new("Max");
dealer.NewCarInfo += max.NewCarIsHere;
dealer.NewCar("Aston Martin");
dealer.NewCarInfo -= sebastian.NewCarIsHere;
dealer.NewCar("Ferrari");
```

When you run the application, a Williams arrives, and Sebastian is informed. After that, Max registers for the subscription as well, and both Sebastian and Max are informed about the new Aston Martin. Then Sebastian unsubscribes, and only Max is informed about the Ferrari:

```
CarDealer, new car Williams
Sebastian: car Williams is new
CarDealer, new car Aston Martin
Sebastian: car Aston Martin is new
```

```
Max: car Aston Martin is new
CarDealer, new car Ferrari
Max: car Ferrari is new
```

SUMMARY

This chapter provided the basics of delegates, lambda expressions, and events. You learned how to declare a delegate and add methods to the delegate list; you learned how to implement methods called by delegates with lambda expressions; and you learned the process of declaring event handlers to respond to an event, as well as how to create a custom event and use the patterns for raising the event.

Using delegates and events in the design of a large application can reduce dependencies and the coupling of layers. This enables you to develop components that have a higher reusability factor.

Lambda expressions are C# language features based on delegates. With these, you can reduce the amount of code you need to write.

The next chapter covers the use of different forms of collections.